Linear logic and lazy computation

J.Y. Girard, Y. Lafont

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 588

LINEAR LOGIC AND LAZY COMPUTATION

Jean-Yves GIRARD
Yves LAFONT

Décembre 1986

Linear Logic and Lazy Computation (Logique Linéaire et Calcul Paresseux)

J.Y. Girard
Equipe de Logique, UA753 du CNRS
Mathématiques, Tour 45-55, 5^e étage
2 place Jussieu, 75251 Paris CEDEX 05

Y. Lafont
INRIA, Projet Formel

Résumé

Récemment, J.Y.Girard a mis en évidence le fait que les connecteurs logiques habituels comme $\Rightarrow$ (l'implication) peuvent être décomposés en connecteurs *linéaires* plus primitifs. D'où une nouvelle logique *linéaire* [Girard86] dans laquelle les hypothèses sont (en un certain sens) utilisées *une fois et une seule*. Le plus étonnant est qu'en ajoutant des opérateurs récursifs (le connecteur "bien sûr"), on retrouve la puissance expressive de la logique habituelle.

Il existe deux versions de la *logique linéaire*: la version *intuitionniste* et la version *classique*. Il semble que la seconde fournit un formalisme adapté aux problèmes de *parallélisme* et de *communication*. Cette approche est entièrement nouvelle et requiert un développement ultérieur. Ici, on se restreint à la version *intuitionniste* et aux conséquences de la contrainte *linéaire* sur le mécanisme de calcul.

On donne deux présentations équivalentes (du fragment propositionnel) de la *logique linéaire intuitionniste*: un calcul de séquents et un système de combinateurs catégoriques.

On introduit les notions de connecteurs *inductifs* et *projectifs*, en particulier le connecteur ! (lire "bien sûr") qui joue un rôle fondamental pour la traduction de la logique intuitionniste habituelle dans la *logique linéaire*.

On a un *théorème d'élimination des coupures* dont la contrepartie pour le système de combinateurs est un *mécanisme d'évaluation*. On présente une machine (virtuelle) très simple pour le calcul *linéaire*, avec les caractéristiques suivantes:

- Un mécanisme d'évaluation paresseux *très naturel*
- Pas besoin de *"garbage collector"*

Enfin, on considère la possibilité d'utiliser la *logique linéaire* pour implanter les langages fonctionnels.

Abstract

Recently, J.Y. Girard discovered that usual logical connectors such as $\Rightarrow$ (implication) could be broken up into more elementary *linear* connectors. This provided a new *linear* logic [Girard86] where hypothesis are (in some sense) used *once* and *only once*. The most surprising is that all the power of the usual logic can be recovered by means of recursive logical operators (connector "of course").

There are two versions of the *linear logic*: the *intuitionistic* one and the *classical* one. It seems that the second provides a appropriate formalism for *parallelism* and *communication*. This approach is entirely new and requires a further development. Here we restrict our attention to the *intuitionistic* version and to the consequences of the *linear* constraint to the computation process.

We give two equivalent presentations of the (propositional part of) *linear* logic: a sequent calculus and a (categorical) combinator system.

Then we introduce *inductive* and *projective* connectors, in particular the connector ! (read "of course"). It plays a fundamental role in the encoding of usual intuitionistic logic into *linear* logic.

There is a *cut elimination theorem* for the sequent calculus that corresponds to an *evaluation mechanism* for the combinator system. We present a very simple (abstract) machine that performs *linear* computations with the following features:

- A *very natural* lazy evaluation mechanism.
- No need of *garbage collector*.

Finally, we discuss the relevance of *linear* logic to implement functional languages.

1 A sequent calculus for the linear intuitionistic logic

First, we present the *elementary* part of the *linear intuitionistic logic* in a 'Gentzen like' formalism [Gentzen].

The connectors are **1** (tensor unit), $\otimes$ (tensor product), **t** (direct unit), & (direct product), **0** (direct zero), $\oplus$ (direct sum) and $\multimap$ (linear implication). Thus there are two different conjunctions (the tensor product and the direct product).

In the following rules A, B, C denote formulas and Γ, Δ denote sequences $A_1, ..., A_n$ of formulas. A *sequent* $A_1, ..., A_n \vdash A$ means that A is a consequent of $A_1 \otimes ... \otimes A_n$.

1.1 Structural rules

$$\frac{}{A \vdash A}\ \text{(identity)} \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\ \text{(cut)} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ \text{(exchange)}$$

1.2 Logical rules

$$\frac{}{\vdash \mathbf{1}} \qquad \frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}$$

$$\frac{}{\Gamma \vdash \mathbf{t}} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \& B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \& B \vdash C}$$

$$\frac{}{\Gamma, \mathbf{0} \vdash A} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}$$

The essential difference with the usual intuitionistic calculus is the *absence* of two *essentially non linear* structural rules:

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{(weakening)} \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{(contraction)}$$

See appendix A for comparison.

Theorem 1 *This calculus admits cut elimination: "every proof without hypothesis can be transformed into a cut free proof".*

The proof is essentially the same as Gentzen's one (for usual intuitionistic logic). It is even simpler because of the absence of weakening and contraction.

Let us remind that the *cut elimination* property has very pleasant consequences: the *consistency*[1] of the system and the *subformula* property (a cut free proof contains only subformulas of the sequent that it proves).

1.3 Examples of proofs

From now on, *Heyting* (logic, formula, proof) means *usual intuitionistic*.

A first (very crude) interpretation of the system is to see **1** and **t** as the *true* proposition, $\otimes$ and & as a conjunction, **0** as the *false* proposition, $\oplus$ as a disjunction and $\multimap$ as an implication. With this translation, every provable *linear* formula becomes obviously a provable *Heyting* formula.

For example, the *linear* formula $(A\&B)\multimap A$ (here A and B are atomic formulas) has the following (*linear*) proof:

$$\dfrac{\dfrac{A \vdash A}{A\&B \vdash A}}{\vdash (A\&B)\multimap A}$$

Of course, the corresponding *Heyting* formula $(A \wedge B) \Rightarrow A$ is also provable.

But the converse is absolutely false: The *linear* formula $(A \otimes B)\multimap A$ is not (*linearly*) provable. Yet the corresponding *Heyting* formula $(A \wedge B) \Rightarrow A$ is still provable.

Let us show that $(A \otimes B)\multimap A$ is not provable: Take a *cut free* proof of $\vdash (A \otimes B)\multimap A$. The end of your proof *has to be*:

$$\dfrac{\dfrac{A, B \vdash A}{A \otimes B \vdash A}}{\vdash (A \otimes B)\multimap A}$$

In *linear* logic, it is impossible to prove $A, B \vdash A$ (in a cut free proof, the last rule has to be an exchange ... and you cannot find a beginning for this proof).

One of the notable features of the *linear* logic is the following distributivity property ($A \equiv B$ means that $A \vdash B$ and $B \vdash A$ are both provable):

$$A \otimes (B \oplus C) \equiv (A \otimes B) \oplus (A \otimes C)$$

Of course it is not true if you replace $\otimes$ by &.

A *cut* free proof for the left to right sense is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A} \quad \dfrac{}{B \vdash B}}{A, B \vdash A \otimes B}}{A, B \vdash (A \otimes B) \oplus (A \otimes C)} \quad \dfrac{\dfrac{\dfrac{}{A \vdash A} \quad \dfrac{}{C \vdash C}}{A, C \vdash A \otimes C}}{A, C \vdash (A \otimes B) \oplus (A \otimes C)}}{\dfrac{A, B \oplus C \vdash (A \otimes B) \oplus (A \otimes C)}{A \otimes (B \oplus C) \vdash (A \otimes B) \oplus (A \otimes C)}}$$

Such a *cut free* proof is easy to find in a bottom up fashion.

---

[1] In our case, the consistency is obvious (it is just a propositional calculus). Moreover there is a very simple translation of the linear logic into the usual one that preserves provability (see section 1.3).

2 Combinators for the linear logic

Linear combinators are an alternative presentation for the linear logic[2].

A combinator is a "name" for an assertion $A \to B$ (B is consequent of A), where A, B are *formulas*. In some sense, combinators are more elementary than sequent rules. Sequent proofs are better for the human, but combinators are closer to the machine.

2.1 Sequential combinators

$$\frac{}{\mathbf{id} : A \to A} \qquad \frac{x : A \to B \quad y : B \to C}{y \circ x : A \to C}$$

2.2 Parallel and arrange combinators

$$\frac{}{\mathbf{1} : \mathbf{1} \to \mathbf{1}} \qquad \frac{x : A \to B \quad y : C \to D}{x \otimes y : A \otimes C \to B \otimes D}$$

$$\frac{}{\mathbf{ol} : A \leftrightarrow \mathbf{1} \otimes A : \mathbf{cl}} \qquad \frac{}{\mathbf{or} : A \leftrightarrow A \otimes \mathbf{1} : \mathbf{cr}}$$

$$\frac{}{\mathbf{ex} : A \otimes B \leftrightarrow B \otimes A : \mathbf{ex}} \qquad \frac{}{\mathbf{al} : A \otimes (B \otimes C) \leftrightarrow (A \otimes B) \otimes C : \mathbf{ar}}$$

2.3 Logical combinators

$$\frac{}{\langle\rangle : X \to \mathbf{t}} \qquad \frac{x : X \to A \quad y : X \to B}{\langle x, y\rangle : X \to A\&B} \qquad \frac{}{\mathbf{fst} : A\&B \to A} \qquad \frac{}{\mathbf{snd} : A\&B \to B}$$

$$\frac{}{\{\} : \mathbf{0} \to X} \qquad \frac{}{\mathbf{inl} : A \to A \oplus B} \qquad \frac{}{\mathbf{inr} : B \to A \oplus B} \qquad \frac{x : A \to X \quad y : B \to X}{\{x, y\} : A \oplus B \to X}$$

$$\frac{x : X \otimes A \to B}{\mathbf{lcur}\, x : X \to A \multimap B} \qquad \frac{}{\mathbf{lapp} : (A \multimap B) \otimes A \to B}$$

For comparison see appendix B.

Proposition 1 *The two formalisms (sequents and combinators) are equivalent:*

Every combinator $\varphi : A \to B$ gives a proof of $A \vdash B$, and every proof of a sequent $A_1, ..., A_n \vdash B$ gives a combinator $\varphi : A_1 \otimes ... \otimes A_n \to B$.

---

[2]They are the exact analogues of what are *categorical combinators* for *Heyting* logic (see appendix B) [Lambek80,Curien85,Curien86]. For a category theoretical view, see appendix C.

The proof is straigthforward. In the following rules, Γ is cumbersome, but you can push it to the right side using the connector $\multimap$:

$$\frac{}{\Gamma, \mathbf{0} \vdash A} \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}$$

3 The connector "of course"

The *linear* constraint is very strong. To recover the expressiveness of *Heyting* logic, it is necessary to introduce a new connector: ! (read *"of course"*).

More generally, we can enrich the *linear* logic with *inductive* and *projective* connectors, two dual notions that we illustrate in the following sections:

3.1 Inductive connectors

Let us construct a "type" of natural numbers in our *linear* logic.

The first solution is a recursive definition (a natural number is *zero* or the *successor* of a natural number):

$$\mathbf{Nat} = \mathbf{1} \oplus \mathbf{Nat}$$

However, this definition does not capture the fact that **Nat** is the "best" solution of this "equation". In particular, you need recursive definitions to construct usual functions over integers.

A more adequate solution is to introduce explicit combinators:

$$\frac{}{\mathbf{zero} : \mathbf{1} \to \mathbf{Nat}} \qquad \frac{}{\mathbf{succ} : \mathbf{Nat} \to \mathbf{Nat}} \qquad \frac{x : \mathbf{1} \to X \quad y : X \to X}{\mathbf{nrec}\, x\, y : \mathbf{Nat} \to X}$$

Let us give, for example, a (non recursive) definition of the addition:

$$\frac{\dfrac{}{\mathbf{cl} : \mathbf{1} \otimes \mathbf{Nat} \to \mathbf{Nat}}}{\mathbf{lcur\ cl} : \mathbf{1} \to \mathbf{Nat} \multimap \mathbf{Nat}}$$

$$\frac{\dfrac{\dfrac{}{\mathbf{lapp} : (\mathbf{Nat} \multimap \mathbf{Nat}) \otimes \mathbf{Nat} \to \mathbf{Nat}} \quad \dfrac{}{\mathbf{succ} : \mathbf{Nat} \to \mathbf{Nat}}}{\mathbf{succ} \circ \mathbf{lapp} : (\mathbf{Nat} \multimap \mathbf{Nat}) \otimes \mathbf{Nat} \to \mathbf{Nat}}}{\mathbf{lcur}(\mathbf{succ} \circ \mathbf{lapp}) : \mathbf{Nat} \multimap \mathbf{Nat} \to \mathbf{Nat} \multimap \mathbf{Nat}}$$

$$\mathbf{add} = \mathbf{lapp} \circ ((\mathbf{nrec}(\mathbf{lcurcl})(\mathbf{lcur}(\mathbf{succ} \circ \mathbf{lapp}))) \otimes \mathbf{id}) : \mathbf{Nat} \otimes \mathbf{Nat} \to \mathbf{Nat}$$

Other *inductive connectors* can be introduced, for example the connector List with the following recursive definition:

$$A\,\mathbf{List} = \mathbf{1} \oplus (A \otimes (A\,\mathbf{List}))$$

The reader may find the corresponding combinators ...

3.2 Projective connectors

If you replace $\oplus$ by &, you obtain the dual notion of *projective connector*.

For example the connector ! has the following recursive definition:

$$!A = A\&\mathbf{1}\&(!A\otimes !A)$$

As for **Nat**, we introduce new combinators:

$$\frac{x : X \to A \quad y : X \to \mathbf{1} \quad z : X \to X \otimes X}{\mathbf{make}\ x\ y\ z : X \to !A}$$

$$\frac{}{\mathbf{read} : !A \to A} \qquad \frac{}{\mathbf{kill} : !A \to \mathbf{1}} \qquad \frac{}{\mathbf{dupl} : !A \to !A \otimes !A}$$

The connector ! is a "trick" to eliminate the *linear* constraint.

First, $!A$ is a "universal coalgebra" over A:

Proposition 2 *The following combinator can be constructed:*

$$\frac{x : !A \to B}{\mathbf{lift}\ x : !A \to !B}$$

! is also a sort of "exponential" operator (it links together the two conjunctions & and $\otimes$):

Proposition 3 $!\mathbf{t} \equiv \mathbf{1}$ *and* $!(A\&B) \equiv !A \otimes !B$

In other words, the following combinators can be constructed:

$$\mathbf{subl} : !\mathbf{t} \leftrightarrow \mathbf{1} : \mathbf{crys} \qquad \mathbf{crac} : !(A\&B) \leftrightarrow !A \otimes !B : \mathbf{glue}$$

See appendix D for detailed constructions.

3.3 Encoding of Heyting logic into linear logic

We saw in section 1.3 a translation of *linear* logic into Heyting logic. Conversely, Heyting logic can be *merged* into *linear* logic (with the connector !).

First we give a translation of *Heyting* formulas into *linear* ones:

- $|A| = A$ (A is an atomic formula)
- $|t| = \mathbf{t}$ and $|A \wedge B| = |A|\&|B|$
- $|f| = \mathbf{0}$ and $|A \vee B| = !|A| \oplus !|B|$
- $|A \Rightarrow B| = !|A| \multimap |B|$

Proposition 4 *A Heyting formula A is provable if and only if $|A|$ is (linearly) provable.*

The proof uses the following lemma[3]:

Lemma 1 *Let A,B be two* Heyting *formulas. Every categorical combinator* $x : A \to B$ *gives a linear combinator:* $|x| : !|A| \to |B|$.

For example, if $x : A \wedge B \to C$ gives $|x| : !(|A| \& |B|) \to C$, then $\mathbf{cur}\, x : A \to B \Rightarrow C$ gives $\mathbf{lcur}(|x| \circ \mathbf{glue}) : !|A| \to !|B| \multimap C$.

In fact, the necessary combinators are exactly those introduced by propositions 2 and 3.

4 Computation

4.1 The evaluation mechanism

Our purpose is to show that the *linear* logic is well-suited for *lazy* evaluation (following the philosophy of [Lafont86]).

Lazy types are:

$$\mathbf{t} \quad A \& B \quad \mathbf{0} \quad A \oplus B \quad A \multimap B$$

Values of *lazy* types are not *computed* but *frozen.* A frozen value is made of a *constructor* and another value, and it is unfrozen by a *destructor.*

Constructors are:

$$\langle\rangle \quad \langle \varphi, \psi \rangle \quad \mathbf{inl} \quad \mathbf{inr} \quad \mathbf{lcur}\, \varphi$$

Destructors are:

$$\mathbf{fst} \quad \mathbf{snd} \quad \{\} \quad \{\varphi, \psi\} \quad \mathbf{lapp}$$

Values are terms:

- $()$
- (u, v) where u, v are values
- $\gamma \cdot u$ where γ is a *constructor* and u a value

We inductively define a relation $u : A$ (u "is a value" of A) for a value u and a *linear* formula A:

$$\frac{}{() : \mathbf{1}} \qquad \frac{u : A \quad v : B}{(u, v) : A \otimes B} \qquad \frac{\gamma : A \to B \quad u : A}{\gamma \cdot u : B}$$

We define an operation: $\dfrac{\varphi : A \to B \quad u : A}{\varphi\, u : B}$

[3]Note that a *Heyting* formula A is provable when there exists a categorical combinator $t \to A$, and a *linear* formula A is (linearly) provable when there exists a *linear* combinator $\mathbf{1} \to A$.

- $\mathbf{id}\, u = u \quad (\varphi \circ \psi)\, u = \varphi(\psi\, u)$
- $\mathbf{1}() = () \quad (\varphi \otimes \psi)(u, v) = (\varphi\, u, \psi\, v)$
- $\mathbf{ol}\, u = ((), u) \quad \mathbf{cl}((), u) = u \quad \mathbf{or}\, u = (u, ()) \quad \mathbf{cr}(u, ()) = u$
  $\mathbf{ex}(u, v) = (v, u) \quad \mathbf{al}(u, (v, w)) = ((u, v), w) \quad \mathbf{ar}((u, v), w) = (u, (v, w))$
- $\langle\rangle\, u = \langle\rangle \cdot u \quad \langle\varphi, \psi\rangle\, u = \langle\varphi, \psi\rangle \cdot u$
  $\mathbf{fst}(\langle\varphi, \psi\rangle \cdot u) = \varphi\, u \quad \mathbf{snd}(\langle\varphi, \psi\rangle \cdot u) = \psi\, u$
- $\mathbf{inl}\, u = \mathbf{inl} \cdot u \quad \mathbf{inr}\, u = \mathbf{inr} \cdot u$
  $\{\varphi, \psi\}(\mathbf{inl} \cdot u) = \varphi\, u \quad \{\varphi, \psi\}(\mathbf{inr} \cdot u) = \psi\, u$
- $(\mathbf{lcur}\, \varphi)\, u = (\mathbf{lcur}\, \varphi) \cdot u \quad \mathbf{lapp}((\mathbf{lcur}\, \varphi) \cdot u, v) = \varphi(u, v)$

Theorem 2 *The previous definition is well founded: Computations using these rules always terminate.*

The proof uses induction over combinators, values and formulas.

The theorem extends to inductive and projective connectors. For example for the connector !:

- $(\mathbf{make}\, \varphi\, \psi\, \rho)\, u = (\mathbf{make}\, \varphi\, \psi\, \rho) \cdot u$
  $\mathbf{read}((\mathbf{make}\, \varphi\, \psi\, \rho) \cdot u) = \varphi\, u$
  $\mathbf{kill}((\mathbf{make}\, \varphi\, \psi\, \rho) \cdot u) = \psi\, u$
  $\mathbf{dupl}((\mathbf{make}\, \varphi\, \psi\, \rho) \cdot u) = ((\mathbf{make}\, \varphi\, \psi\, \rho) \otimes (\mathbf{make}\, \varphi\, \psi\, \rho))(\rho\, u)$

Finally, we may add primitive types with primitive values and primitive combinators, for example a type **Num** with:

- $\mathbf{0} : \mathbf{Num} \quad \mathbf{1} : \mathbf{Num} \quad \mathbf{2} : \mathbf{Num} \ldots$
  $\mathbf{minus} : \mathbf{Num} \to \mathbf{Num} \quad \mathbf{sum} : \mathbf{Num} \otimes \mathbf{Num} \to \mathbf{Num} \ldots$

4.2 The Linear Abstract Machine

The *Linear Abstract Machine* is a cousin of the *Categorical Abstract Machine* [CAM]. But the *linear* constraint allows a radically different allocation of the memory space.

The memory space is divided into three areas:

- The *code* area is *static* (the code doesn't change during the execution) and is organized as a graph.
- The *environment* area is *dynamic* with two part: the *current tree* (or actual environment) and the *free list* (or memory heap).
- The *dump* (or stack) is *dynamic* and *linear*.

The main point is that the actual environment is organized as a tree[4], and the space allocation is completely provided (no need of *garbage collector*, see section 5).

As usual, the code is a list of elementary instructions (notations: $c :: C$ denotes the list whose head is c and whose tail is C, [] denotes the empty list and γ denotes a constructor).

Linear Abstract Machine					
Before			After		
code	environment	dump	code	environment	dump
$pushl :: C$	(u,v)	D	C	u	$v :: D$
$consl :: C$	u	$v :: D$	C	(u,v)	D
$pushr :: C$	(u,v)	D	C	v	$u :: D$
$consr :: C$	v	$u :: D$	C	(u,v)	D
$ol :: C$	u	D	C	$((),u)$	D
$cl :: C$	$((),u)$	D	C	u	D
$or :: C$	u	D	C	$(u,())$	D
$cr :: C$	$(u,())$	D	C	u	D
$ex :: C$	(u,v)	D	C	(v,u)	D
$al :: C$	$(u,(v,w))$	D	C	$((u,v),w)$	D
$ar :: C$	$((u,v),w)$	D	C	$(u,(v,w))$	D
$\gamma :: C$	u	D	C	$\gamma \cdot u$	D
$fst :: C$	$(pair(C',C'')) \cdot u$	D	C'	u	$C :: D$
$snd :: C$	$(pair(C',C'')) \cdot u$	D	C''	u	$C :: D$
$altv(C',C'') :: C$	$inl \cdot u$	D	C'	u	$C :: D$
$altv(C',C'') :: C$	$inr \cdot u$	D	C''	u	$C :: D$
$lapp :: C$	$((lcurC') \cdot u, v)$	D	C'	(u,v)	$C :: D$
[]	u	$C :: D$	C	u	D
[]	u	[]		Return u	

Every *linear* combinator φ gives code $\|\varphi\|$ for the LAM (notation: @ denotes the concatenation of lists):

- $\|\mathbf{id}\| = []$ $\quad \|\varphi \circ \psi\| = \|\psi\| @ \|\varphi\|$
- $\|\mathbf{1}\| = \|\mathbf{id}\| = []$
  $\|\varphi \otimes \psi\| = \|(\mathbf{id} \otimes \psi) \circ (\varphi \otimes \mathbf{id})\| = [pushl] @ \|\varphi\| @ [consl; pushr] @ \|\psi\| @ [consr]$

For the other connectors, the translation is obvious:

- $\|\langle \varphi, \psi \rangle\| = [pair(\|\varphi\|, \|\psi\|)]$ $\quad \|\mathbf{fst}\| = [fst]$ ...

4.3 Compilation of inductive and projective combinators

There is no specific LAM instruction for inductive and projective combinators. In fact they can be compiled into looping code.

Let us consider for example the connector !, with its recursive definition:

[4] In a strong sense, that means a connected graph without cycle and without shared nodes.

$$!A = A\&\mathbf{1}\&(!A \otimes !A)$$

!A is a direct product with three projections (Here, **trd** denotes the third projection):

$$\mathbf{read} = \mathbf{fst} : !A \to A \qquad \mathbf{kill} = \mathbf{snd} : !A \to \mathbf{1} \qquad \mathbf{dupl} = \mathbf{trd} : !A \to !A \otimes !A$$

The combinator **make** is compiled into the following *looping* combinator:
$\mathbf{make}\, x\, y\, z = m$ where $m = \langle x, y, (m \otimes m) \circ z \rangle$

5 Relevance of linear logic for computation

5.1 Lazyness

We have to clarify the difference between *Heyting* logic and *linear* logic, and the simplification *linear* logic gives.

In *Heyting* logic, there is only one conjunction $\wedge$:

A *strict* value of $A \wedge B$ is a pair (u, v) where u is a value of A and v a value of B. Such a value may be too "evaluated" if you apply the destructor **fst** or **snd**.

A *lazy* value of $A \wedge B$ is a frozen value $\langle \varphi, \psi \rangle \cdot u$ where u is a value of a type X and φ, ψ are combinators, $\varphi : X \to A$ and $\psi : X \to B$. Such a value may be too little "evaluated" if you apply the destructor **app**.

Of course, it is possible to *unfreeze* frozen values when necessary, but this mechanism seems rather complicated and unnatural, compared to the strict evaluation mechanism [CAM,MaSu].

The problem is that two essentially different kinds of destructors (the *projections* and the *application*) may operate over values of type $(A \Rightarrow B) \wedge A$.

In *linear* logic, the dilemma disappears:

Values of $A \otimes B$ are *strict* values, and the two components are necessary: There is no projection $A \otimes B \to A$ or $A \otimes B \to B$.

Values of $A\&B$ are *lazy* values, and the only possible destructors for such a value are fst and **snd**.

5.2 Memory allocation

Implementations of symbolic (LISP) or functional (ML) languages need a separate mechanism (the *garbage collector*) to recover the memory space used by abandoned pieces of data. Garbage collecting takes time and sometimes place. Moreover, it complicates the implementation ...

In *linear* logic, the connector corresponding to the management of environment is $\otimes$ ($\&$ is *lazy*). Thus, *projections* and *pairing* don't act on the environment. This allows the environment to be kept in a tree whose nodes are never *abandoned* or *shared*.

More precisely, the transitions of the *Linear Abstract Machine* are *left* and *right* linear with respect to the environment (but not to the code). *Left* linearity is expected for an abstract machine, but *right* linearity is rather surprising.

In our "implementation", we add a fourth register (the *free list*) to the *Linear Abstract Machine*. Some instructions (*consl*, *consr*, *ol*, *or* and the constructors) *take* a free location from the *free list*. Other instructions (*pushl*, *pushr*, *cl*, *cr*, *fst*, *snd*, *altv*, *lapp*) return a location to the *free list* (this is legitimate because nodes are not shared). The other instructions (like *ex*) act as physical modifications.

Of course, we don't need a *garbage collector* because nodes are never abandoned.

5.3 Compilation of functional languages

We saw in section 3.3 a translation of categorical combinators into linear combinators. But there is a classical translation of functional programs into categorical combinators [CAM,MaSu]. That gives a compilation of functional programs into the *Linear Abstract Machine*.

Unfortunately, this compilation is not realistic. In fact, a very simple program gives a big piece of code. For example, the categorical combinator $\varphi \circ \psi$ is translated into the linear combinator $|\varphi| \circ (\mathbf{lift}\ |\psi|)$, and **lift** is not a primitive combinator (see appendix D), and **make** is not a primitive instruction (see section 4.3) ...

Of course, this translation is too brutish. The problem is to understand how the *linearity* that occurs in a program (and there is a lot of linearity in classical algorithms) can be recognized (by the machine or by the programmer) for our *linear* implementation.

A possible continuation for this article would be the elaboration of a realistic optimized translation, or rather the development of a new programming style, in a new high level language adapted to our *linear* implementation. This new language would hold simultaneously elegance of functional languages and efficiency of procedural languages.

Appendix

A The sequent calculus for the usual intuitionistic logic

A.1 Structural rules

$$\frac{}{A \vdash A}\ \text{(identity)} \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\ \text{(cut)} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ \text{(exchange)}$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \text{(weakening)} \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{(contraction)}$$

A.2 Logical rules

$$\frac{}{\vdash t} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C}$$

$$\frac{}{f \vdash A} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma, A \vdash C \quad \Delta, B \vdash C}{\Gamma, \Delta, A \vee B \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \Rightarrow B \vdash C}$$

B Categorical combinators

B.1 Sequential combinators

$$\frac{}{\mathrm{id} : A \to A} \qquad \frac{x : A \to B \quad y : B \to C}{y \circ x : A \to C}$$

B.2 Logical combinators

$$\frac{}{\langle\rangle : X \to t} \qquad \frac{x : X \to A \quad y : X \to B}{\langle x, y \rangle : X \to A \wedge B} \qquad \frac{}{\mathrm{fst} : A \wedge B \to A} \qquad \frac{}{\mathrm{snd} : A \wedge B \to B}$$

$$\frac{}{\{\} : f \to X} \qquad \frac{}{\mathrm{inl} : A \to A \vee B} \qquad \frac{}{\mathrm{inr} : B \to A \vee B} \qquad \frac{x : A \to X \quad y : B \to X}{\{x, y\} : A \vee B \to X}$$

$$\frac{x : X \wedge A \to B}{\mathrm{cur}\, x : X \to A \Rightarrow B} \qquad \frac{}{\mathrm{app} : (A \Rightarrow B) \wedge A \to B}$$

C Linear categories

C.1 Terminology

A *symmetric monoidal* category is a category C with a bifunctor $\otimes : C \times C \to C$ and an object $\mathbf{1} \in C$ such that:

$$\mathbf{1} \otimes X \overset{\nu}{\cong} X \qquad X \otimes Y \overset{\sigma}{\cong} Y \otimes X \qquad (X \otimes Y) \otimes Z \overset{\alpha}{\cong} X \otimes (Y \otimes Z)$$

Here $\cong$ denotes a *natural isomorphism.* In addition, there are several *coherence* axioms that constrain those natural isomorphims (for example: $\sigma \circ \sigma = \mathbf{id}$, $\nu \circ \alpha = \nu \otimes \mathbf{id}$).

A *symmetric monoidal closed* category is a symmetric monoidal category C such that, for every $A \in C$, the functor $X \mapsto X \otimes A$ has a right adjoint $Y \mapsto A \multimap Y$. That means:

$$\mathbf{Hom}(X \otimes A, Y) \cong \mathbf{Hom}(X, A \multimap Y)$$

Finally, a *linear* category is a symmetric monoidal closed category with finite products and coproducts[5].

C.2 Examples

Of course, a category with finite products (or coproducts) is a monoidal category. Therefore a category with finite products, finite coproducts and exponentials is a linear category[6]. For example **SET** is a linear category: $\otimes$ and & are the cartesian product, $\oplus$ is the disjoint union, and $I \multimap J = J^I$.

A more interesting model is the category of modules over a ring: $\otimes$ is the tensor product, & the direct product, $\oplus$ the direct sum, and $A \multimap B = \mathbf{Hom}(A, B)$. Of course, & and $\oplus$ are identical.

Another example is the category **TOP** of topological spaces. **TOP** is not cartesian closed but it is a linear category: & is the cartesian product and $\oplus$ is the disjoint union. $E \otimes F$ is $E \times F$ with the finest topology that makes sections $x \mapsto (x, y)$ and $y \mapsto (x, y)$ continuous. $E \multimap F$ is the space of continuous maps $E \to F$ with the pointwise convergence topology.

D Some useful linear combinators

$$\mathbf{trans} = \mathbf{ar} \circ ((\mathbf{al} \circ (\mathbf{id} \otimes \mathbf{ex}) \circ \mathbf{ar}) \otimes \mathbf{id}) \circ \mathbf{al} : (A \otimes B) \otimes (C \otimes D) \to (A \otimes C) \otimes (B \otimes D)$$

$$\frac{x : !A \to B \quad \overline{\mathbf{kill} : !A \to \mathbf{1}} \quad \overline{\mathbf{dupl} : !A \to !A \otimes !A}}{\mathbf{lift}\ x = \mathbf{make}\ x\ \mathbf{kill}\ \mathbf{dupl} : !A \to !B}$$

[5]The categorical notion corresponding to "!" is more complex. It makes use of the notion of *internal comonoïd.*

[6]This justifies the translation of section 1.3. Moreover, in such a category, "!" exists (it's the identity functor).

[Gentzen] G. Gentzen. "The Collected Papers of Gerhard Gentzen." Ed. E. Szabo, North-Holland, Amsterdam (1969).

[Girard86] J.Y. Girard "Linear Logic" to appear in TCS.

[Lafont86] Y. Lafont "De la Déduction Naturelle à Machine Catégorique" to appear.

[Lambek80] J. Lambek. "From Lambda-calculus to Cartesian Closed Categories." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[MaSu] M. Mauny and A. Suarez. "Implementing Functional Languages in the Categorical Abstract Machine." in Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

$$\frac{}{\mathbf{subl} = \mathbf{kill} : !\mathbf{t} \to \mathbf{1}}$$

$$\frac{\overline{\langle\rangle : \mathbf{1} \to \mathbf{t}} \quad \overline{\mathbf{id} : \mathbf{1} \to \mathbf{1}} \quad \overline{\mathbf{ol} : \mathbf{1} \to (\mathbf{1} \otimes \mathbf{1})}}{\mathbf{crys} = \mathbf{make}\ \langle\rangle\ \mathbf{id}\ \mathbf{ol} : \mathbf{1} \to !\mathbf{t}}$$

$$\frac{\dfrac{\overline{\mathbf{fst} : A\&B \to A}}{!\mathbf{fst} : !(A\&B) \to !A} \quad \dfrac{\overline{\mathbf{snd} : A\&B \to B}}{!\mathbf{snd} : !(A\&B) \to !B}}{!\mathbf{fst} \otimes !\mathbf{snd} : !(A\&B) \otimes !(A\&B) \to !A \otimes !B}$$

$$\mathbf{crac} = (!\mathbf{fst} \otimes !\mathbf{snd}) \circ \mathbf{dupl} : !(A\&B) \to !A \otimes !B$$

$$\frac{\dfrac{\overline{\mathbf{read} : !A \to A} \quad \overline{\mathbf{kill} : !B \to \mathbf{1}}}{\mathbf{read} \otimes \mathbf{kill} : !A \otimes !B \to A \otimes \mathbf{1}} \quad \dfrac{}{\mathbf{cr} : A \otimes \mathbf{1} \to A}}{\mathbf{cr} \circ (\mathbf{read} \otimes \mathbf{kill}) : !A \otimes !B \to A}$$

$$\frac{\dfrac{\overline{\mathbf{kill} : !A \to \mathbf{1}} \quad \overline{\mathbf{read} : !B \to B}}{\mathbf{kill} \otimes \mathbf{read} : !A \otimes !B \to \mathbf{1} \otimes B} \quad \dfrac{}{\mathbf{cl} : \mathbf{1} \otimes B \to B}}{\mathbf{cl} \circ (\mathbf{kill} \otimes \mathbf{read}) : !A \otimes !B \to B}$$

$$\frac{\dfrac{\overline{\mathbf{kill} : !A \to \mathbf{1}} \quad \overline{\mathbf{kill} : !B \to \mathbf{1}}}{\mathbf{kill} \otimes \mathbf{kill} : !A \otimes !B \to \mathbf{1} \otimes \mathbf{1}} \quad \dfrac{}{\mathbf{cl} : \mathbf{1} \otimes \mathbf{1} \to \mathbf{1}}}{\mathbf{cl} \circ (\mathbf{kill} \otimes \mathbf{kill}) : !A \otimes !B \to \mathbf{1}}$$

$$\frac{\overline{\mathbf{dupl} : !A \to !A \otimes !A} \quad \overline{\mathbf{dupl} : !B \to !B \otimes !B}}{\mathbf{dupl} \otimes \mathbf{dupl} : !A \otimes !B \to (!A \otimes !A) \otimes (!B \otimes !B)}$$

$\mathbf{glue} = \mathbf{make}\ \langle \mathbf{cr} \circ (\mathbf{read} \otimes \mathbf{kill}), \mathbf{cl} \circ (\mathbf{kill} \otimes \mathbf{read}) \rangle\ (\mathbf{cl} \circ (\mathbf{kill} \otimes \mathbf{kill}))\ (\mathbf{trans} \circ (\mathbf{dupl} \otimes \mathbf{dupl})) : !A \otimes !B \to !(A\&B)$

References

[CAM] G. Cousineau, P.L. Curien and M. Mauny. "The Categorical Abstract Machine." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 50–64.

[Curien85] P. L. Curien. "Categorical Combinatory Logic." ICALP 85, Nafplion, Springer-Verlag LNCS 194 (1985).

[Curien86] P. L. Curien. "Categorical Combinators, Sequential Algorithms and Functional Programming." Pitman (1986).